Specification for the UCI interface library

Pablo Sanchez

Contents

1. Introduction

1.1 Purpose

The purpose of this document is to detail the specification of the library `uci-interface`.

1.2 Scope

`uci-interface` is a library that should allow the user to in an simple way (just implementing some virtual functions), to interface his chess engine with the UCI protocol, enabling it to be used with most moder GUIs.

1.3 Definitions, acronyms, and abbreviations

UCI Universal Chess Interface

GUI Graphical User Interface

1.4 References

The specification of the UCI can be found (from be root of the project) at:

`./docs/uci-protocol/uci-spec.rst`

1.5 Overview

- 2. Overall description: Explains the general factors that affect the library and its requirements.
- 3. Specific requirements: Details the public API of the library.

2. Overall description

2.1 Product perspective

To the best of my knowledge, there are no similar libraries that implement this functionality.

2.2 Product functions

By implementing a series of virtual functions, (and optionally using some other utilities from the library), the user should be able to give its chess engine an interface that is fully compliant with the UCI protocol.

2.3 User characteristics

A `C++` developer that wants to implement a chess engine that could be used with most modern GUIs, but does not want to bother with the details of the UCI protocol.

2.4 Assumptions and dependencies

The library is done using only standard `C++`, so it should be completely cross platform.

3. Specific requirements

3.1 External interfaces

The interface with the GUI should be done using `stdin` and `stdout`.

3.2 Public API

Note: Everything is contained in `namespace uci`.

3.2.1 Runtime configuration

You can use `uci::config` to indicate what configuration values can be changed by the GUI, and read those options (See `option` and `setoption` on the UCI spec).

`uci::config` is a `std::unordered_map<const char *, uci::option>`. The way to list those options will be described latter in 3.2.5 Creating the interface.

3.2.1.1 uci::option

Holds the type and string representation of the configuration options.

The types could be any of:

- **check** Check box that could either be true or false.
- **spin** A spin wheel that can be an integer in certain range.
- **combo** A combo box that could have predefined strings as a value.
- **button** A button that can be pressed to send a command to the engine.
- **string** A text field

Note: The button type cant be read from. There will be more details latter, but in short, the button option takes a callback function, and when the GUI sends `setoption name your button`, it will call the callback.

You can use the member function:

```
template <class Type>
typename Type::type option::as(void);
```

To get the value of it.

For example, you could do something like:

```
using uci::option::check;
using uci::option::combo;

bool own_book = uci::config.at("OwnBook").as<check>();
std::string style = uci::config.at("Style").as<combo>();
```

Note: The values given by the config options will always be inside the constraints given. If the GUI sends a `setoption` command with incorrect values, it will be sent an `info` command indicating whats wrong.

3.2.2 Constraints on the moves

The struct `uci::limits` is used to tell the engine the limitations that the GUI will apply to to the calculation of the best move.

The best example of this is when the user wants to play with time control, where the engine will have to take into account the time it has left on the clock.

The members or `uci::limits` are:

- **`std::vector<std::string> search_moves`** Restrict the search to this moves.
- **`bool ponder`** Search in ponder mode.
- **`std::chrono::milliseconds wtime`** The time white has left on the clock (`0` if there are no time needs).
- **`std::chrono::milliseconds btime`** The time black has left on the clock (`0` if there are no time needs).
- **`std::chrono::milliseconds winc`** The increment that white has.
- **`std::chrono::milliseconds binc`** The increment that black has
- **`size_t moves_to_go`** The amount of moves till the next time control. (`0` if there is no time control)
- **`size_t depth`** The limit depth that the engine can search (`0` if there is no limit).
- **`size_t nodes`** The amount of nodes that can be searched (`0` if there is no limit).
- **`size_t mate`** Search for mate in `mate` moves (`0` if there is no limit).

- `std::chrono::milliseconds move_time` Search exactly `move_time` milliseconds (`0` if there is no limit from the gui).
- `bool infinite` Search until the stop command. Don't exit the search without being told so.

3.2.3 Sending messages to the GUI

`namespace info` contains optional utilities that you could use to send information to the GUI.

3.2.4.1 Message types

See the UCI protocol specification for greater detail on every one of the following:

- `info::depth`: Used to represent the current depth of the search.
- `info::selective_depth`: Used to represent the current selective depth of the search.
- `info::time`: The time searched in `std::chrono::milliseconds`.
- `info::nodes`: The nodes searched.
- `info::pv`: A list of moves in UCI with the current top line.
- `info::multipv`: For engines that support multipv mode.
- `info::score`: Indented to use one of the nested classes.
  - `info::score::centipawns`: The score in centipawns from the engines point of view
  - `info::score::mate`: Has found mate
  - `info::score::lowerbound`: The score is just a lower bound
  - `info::score::upperbound`: The score is just an upper bound
- `info::current_move`: Currently searching this move
- `info::current_move_number`: Currently searching move number x.
- `info::hashfull`: The hash is x per mill full.
- `info::nodes_per_second`: The nodes per second that are searched.
- `info::table_base_hits`: The number of positions that where found in the endgame table bases.
- `info::shredder_base_hits`: The number of positions that where found in shredder endgame databases.
- `info::cpu_load`: The CPU usage of the engine.
- `info::string`: A `std::string` as a message to be sent. (This can only be the last one to be sent).
- `info::cstring`: The same as `info::string` but using a `const char*`.
- `info::debug`: The same as `info::string` but only logged in debug mode.
- `info::cdebug`: The same as `info::cstring` but only logged in debug mode.
- `info::refutation`: The details of how a move is refuted.
- `info::current_line`: The current line the engine is calculating.

3.2.3.2 Functions

```
template <class MessageType, class ...Arg>
void info::log(MessageType m, Arg ...args);
```

Will log in a UCI info message the information that it is given.

3.2.3.3 Example

Send current best line:

```
// Calculate the best move

using namespace uci;

info::log(
    info::depth{move_tree.depth()},
    info::score::centipawns{move_tree.top_line().eval()},
    info::pv{move_tree.top_line().uci_string()};
);
```

Send debug messages:

```
// Initialize

using namespace uci;

info::log(
    info::cdebug{"Finished initialization"}
);
```

3.2.4 Global variables

`uci::debug` A `std::atomic<bool>` that is used to check if the engine is in debug mode. It will be used internally by the `uci-interface` to check if it should send the `info::debug` and `info::cdebug` messages. The engine is free to use it in case it needs to do extra checks in debug mode.

`uci::stop_searching` A `std::atomic<bool>` that is used to tell the engine that it should stop searching.

3.2.5 Creating the interface

To create the interface, you should inherit from `uci::engine_interface`, and implement the following virtual functions.

- `bool check_register(void)` Return `true` if the automatic register check was successful. If for the register check you need the user and code, only implement the next function. If you don't have to check for registration, implement none.

- `bool check_register(const std::string& user, const std::string& code)` Return `true` if the registration check was successful. **Note:** If registration fails, then the interface will ignore commands until the registration is successful or it receives a `quit` command.
- `bool check_copy_protection(void)` Return `true` if there aren't any copy protection problems. If your engine does not have copy protection don't implement it. **Note:** If the check fails, then the interface will ignore commands until it receives a `quit` command.
- `bool load_options(void)` Load the default options in `uci::config`, and fill the information about the engine. To load the meta data you can use the following functions:
  - `void set_author_name(const char* name)`
  - `void set_engine_name(cosnt char* name)`
  - `void requires_registration(bool v)`
  - `void requires_copy_protection(bool v)`
  - `void can_ponder(bool v)`
- `void update_position(const std::string& fen, const std::string& moves)` Should update the position that the engine holds.
- `std::string get_best_move(uci::limits l)` Should return the best move in UCI format. **Note:** This function will run in another thread.
- `bool ponder_mode(void)` Make the engine run in ponder mode.
- `bool search_mode(void)` Make the engine run in search mode.

3.2.6 Example

```
#include <iostream>

int main(void) {
    std::cout << "Hello world\n";
    return 0;
}
```